A beginning user manual for clj-peg

Richard Lyman
www.lithinos.com

This documentation was written for version 0.6

This manual is written for those who are comfortable developing in Clojure, and who may or may not be comfortable with parsing, lexing, or grammar construction. You should **not** need to have an advanced degree to understand this manual.

I have written each section as tightly as I could, providing for shorter and more numerous sections. If you're unsure where to start, simply start reading from the very beginning and continue. Each section will flow into the next and should progress nicely. If there are parts where my explanations leave you confused, please let me know.

If you're familiar with terms such as AST or non-greedy consumption you should skip along on the section titles and skim the code until you find something new. If there are parts that you find ambiguous, please let me know.

Either way, I hope this manual helps you to get up to speed as quickly as you want. I also hope that you become effective in using the clj-peg library and that that effectiveness will encourage you to share your PEGs with others.

Overview

When you create a PEG you create a set of functions that are able to parse input. The specific pattern of input that those functions parse is defined in the rules of the PEG you create[1]. Starting with a PEG as defined in Listing 1, we can use the functions it creates to process the input "bbb" as a String. This PEG accepts a string with one or more b's and nothing else.

```
(make-parser {
    :main   simple-parser
    :doc    "This is a simple grammar."
    :rules (
        A <- [(+ B) $]
        B <- "b")})
```

Listing 1: A simple PEG defined in clj-peg

When the newly created functions have finished processing the input "bbb", they return a structure that describes the input, and this is traditionally referred to as an AST (Abstract Syntax Tree). This result of parsing the input is shown in Listing 2. You can see that the data structures used in the result closely match the data structures used in the grammar.

The way the macro `make-parser` works allows you to treat a grammar definition like any other data structure. If you share your grammar, I'd recommend simply tying it to some symbol using `def`, and allowing those who use it to `eval` the related symbol. This gives your users the opportunity of tweaking your grammar to fit their needs. There are more details about the process of tweaking a grammar in later sections.

[1] Note that I'm making a distinction between input pattern and input type. While the pattern of input that can be processed is defined as shown in Listing 1, the type is not necessarily defined. Specifically, clj-peg doesn't care what type of input you provide since it must be wrapped in a specific type of struct. That specific type of struct acts as an interface between the generated parsing code and some input type.

```
{:A
    [
        (
            {:B "b"}
            {:B "b"}
            {:B "b"}
        )
        {:Empty true}
    ]
}
```

Listing 2: The result of applying the parser generated from the grammar in Listing 1 on the input "bbb"

Setup

To start getting our hands dirty with clj-peg we'll need to install it. There really isn't much to installing clj-peg, just download it[2], extract it, and put the JAR file on your classpath. For a more complete walkthrough on setting up a Clojure project, I wrote a post with details on my typical setup. Using that typical setup, I'd place the clj-peg.jar file in the `dst/lib` folder.

After the clj-peg JAR is on the classpath, you can use the library in several ways. For this manual, we'll be including code through the `:use` reference in the `ns` macro. Our main file will look like Listing 3 to begin with.

```
(ns com.company.project.main
   (:gen-class)
   (:use (com.lithinos.clj-peg
                  core
                  string-peg)))

(defn -main [& args]
    (println "Works"))
```

Listing 3: Using clj-peg code

For help with the commands to compile and run a Clojure application, I'd again refer you to my post with details on my typical setup that includes very simple Ant scripts. You should easily be able to understand what needs to happen when compiling or running a Clojure application, and should easily be able to integrate that process into your own build approach if you choose to not use those Ant scripts.

Examples

I'm going to provide some examples in the following sections, but there are a few explanations that come first. In any clj-peg definition, the three items to notice are the main function, the doc string, and the rules. The main function is set to `math`, which is what you'll call to use the parser. The doc string is what accompanies the resulting main function. The rules, commonly referred to as productions, define the different non-terminals and terminals as well as how they're joined to form all of the possible combinations of input that can be parsed.

[2] Depending on your intended use, you may need to purchase a license.

Sequence	**[A B]**	The parser must consume A first, and that must be followed by consuming B.
Ordered choice	**(| A B)**	The parser must either consume A or B, and the parser must try A first, followed by trying B.
Zero or more	**(* A)**	The parser must greedily consume zero or more of A.
One or more	**(+ A)**	The parser must greedily consume at least one, and will greedily consume more than one of A if it can.
One or none	**(? A)**	The parser must consume one of A if it can.
And predicate	**(& A)**	The parser must be able to consume A and must *not* consume A.
Not predicate	**(! A)**	The parser must not be able to consume A.

Table 1: Operators and delimiters for combining non-terminals and terminals in a clj-peg definition.

Rules, in clj-peg, use operators and delimiters to combine non-terminals. If you are familiar with PEG grammars in general, you'll find that these fit the basic definition. All of the provided combinations are shown in Table 1. These delimiters and operators follow Clojure data structures and function calls.

Simple Example

We're going to be developing a grammar that parses simple mathematical operations. For now we'll only support positive whole integers as operands, and we'll only support addition and subtraction as operators. The entire result[3] can be seen in Listing 5, so let's look at the grammar tied to the symbol `g`.

In this specific definition, the non-terminals are `Expr`, `Sum`, and `Value` and the terminals are `Num`, and `SumOp`. Notice use of the Clojure reader macro for Patterns in the terminals `Num`, and `SumOp`. Currently, non-terminals can be anything that is seqable but is not a String, the rest can be used in terminals.

Writing an interpreter for this simple grammar is fairly easy. Each non-terminal that is part of an AST has the form `{:Non-Terminal <body>}`. There is only one key/value pair in the map, the key will be based on the non-terminal, and the body will match the grammar in structure and use of RHS combinations. Let's look at the result from the grammar in Listing 5 and apply it to the expression "1+2-3". The resulting AST is shown in Listing 4.

Interpreting this AST can be approached by defining a function for selected non-terminals. We'll define functions for the non-terminal mappings `Expr`, `Sum`, and `Value`. We won't bother with Empty, since, in this case, it doesn't matter. The non-terminals `SumOp` and `Num` are easily handled inside `Value` and `Sum`. The complete code is shown in Listing 5.

[3] If we were to distribute this code we would not use `:gen-class`, or have a `main` method, or call `eval` on the grammar.

```
{:Expr [
   {:Sum [
      {:Value {:Num "1"}}
      (
         [{:SumOp "+"} {:Value {:Num "2"}}]
         [{:SumOp "-"} {:Value {:Num "3"}}]
      )
   ]}
   {:Empty true}
]}
```

Listing 4: The AST from the grammar in Listing 5 applied to "1+2-3"

```
(ns com.company.project.main
  (:gen-class)
  (:use (com.lithinos.clj-peg
                    core
                    string-peg)))

(def g '(make-parser {
    :main math
    :doc "The characters +, and - are the only
            operators, and positive whole integers
            are the only operands."
    :rules (
        Expr  <- [Sum $]
        Sum   <- [Value (* [SumOp Value])]
        Value <- (| Num Expr)
        Num   <- #"^[0-9]+"
        SumOp <- #"^[+-]")}))

(declare Expr-interpreter)

(defn Value-interpreter [ast]
  (let [ast (ast :Value)]
    (if (contains? ast :Expr)
      (Expr-interpreter ast)
      (Integer/parseInt (ast :Num)))))

(defn Sum-interpreter [ast]
  (let [ast (ast :Sum)]
    (reduce
      (fn [a b]
        (let [rater  ((first b) :SumOp)
              rand  (Value-interpreter (second b))]
          (if (= rater "+")
            (+ a rand)
            (- a rand))))
      (Value-interpreter (first ast))
      (second ast))))

(defn Expr-interpreter [ast]
  (Sum-interpreter (first (ast :Expr))))

(eval g)

(defn -main [& args]
  (let [input "1+2-3"
        valid-struct (create-string-input input)
        ast       (math valid-struct)
        answer     (Expr-interpreter ast)]
    (println input "=" answer)))
```

Listing 5: A simple example

Advanced Example

Let's extend the simple example with a few changes. We'll be allowing multiplication, and division, and we'll do so in a way that preserves the required precedence. There isn't anything too difficult with these changes.

We'll also be expanding the definition of what numbers are allowed by re-using part of a JSON grammar. This change is important, since it can show how simple it is to mix and match parts of grammars. You can notice that the changes to re-use part of a JSON grammar are even easier than the changes to allow multiplication and division.

```
(ns com.company.project.main
  (:gen-class)
  (:use (com.lithinos.clj-peg core string-peg
    json-parser json-interpreter)))

(def math-grammar '(make-parser {
  :main math
  :doc "The characters +, -, *, and / are the
     only operators, and JSONNumbers are
     the only operands."
  :rules (
    Expr      <- [Sum $]
    Sum       <- [Product (* [SumOp Product])]
    Product   <- [Value (* [ProductOp Value])]
    Value     <- (| Num Expr)
    Num       <- JSONNumber
    SumOp     <- #"^[+-]"
    ProductOp <- #"^[*/]")}))

(declare Expr-interpreter)

(defn Num-interpreter [ast]
  (JSONNumber-interpreter (ast :Num)))

(defn Value-interpreter [ast]
  (let [ast (ast :Value)]
    (if (contains? ast :Expr)
      (Expr-interpreter ast)
      (Num-interpreter ast))))

(defn product-reduction [a b]
  (let [temp-op  ((first b) :ProductOp)
     operator (if (= temp-op "*") * /)
     operand1 a
     operand2 (Value-interpreter (second b))]
    (operator operand1 operand2)))

(defn Product-interpreter [ast]
  (let [ast (ast :Product)]
    (if (contains? (second ast) :Zero-or-more)
      (Value-interpreter (first ast))
      (reduce
        product-reduction
        (Value-interpreter (first ast))
        (second ast)))))

(defn sum-reduction [a b]
  (let [temp-op  ((first b) :SumOp)
     operator (if (= temp-op "+") + -)
     operand1 a
     operand2 (Product-interpreter (second b))]
    (operator operand1 operand2)))

(defn Sum-interpreter [ast]
  (let [ast (ast :Sum)]
    (if (contains? (second ast) :Zero-or-more)
      (Product-interpreter (first ast))
      (reduce
        sum-reduction
        (Product-interpreter (first ast))
        (second ast)))))

(defn math-interpreter [ast]
  (Sum-interpreter (first (ast :Expr))))

(eval json-grammar)
(eval math-grammar)

(defn -main [& args]
  (let [input "1.2*3.4+5"
      valid-struct (create-string-input input)
      ast     (math valid-struct)
      answer    (math-interpreter ast)]
    (println input "=" answer)))
```

Listing 6: An advanced example

The additions for multiplication and division are simple. The non-terminal `Product` becomes similar to what `Sum` was, and placing `Product` below `Sum` in the AST allows our bottom-up interpreter the opportunity of solving products before sums. You can see, as well, that the interpreter for the new `Product` non-terminal is very similar to the interpreter for `Sum`.

The interesting part, however, is the extension of our math grammar to include JSONNumbers. Even though the implementation of the JSONNumber interpreter is not complete, our simple use of it in this case will still work. For the first step, we need to import the `json-interpreter` and `json-parser` namespaces. Second, we reference the `JSONNumber` non-terminal in our math grammar. Third we evaluate the `json-grammar` before we evaluate our grammar, so that the definition for `JSONNumber` is available when we evaluate our grammar. Fourth, and finally, we call the JSONNumber-interpreter inside our Num-interpreter.

Conclusion

I need to write another manual. This one does not cover everything. You've likely noticed that there were things that I didn't explain, such as the create-string-input function. In addition, there are several parts of the clj-peg library that aren't finalized in my mind. These will all require at least one more manual.

For now, this manual is sufficient to scratch my documentation itch. I have too many projects all running along at the same time, and this is just one of them. It was bumped to the front of the queue since I'm using it in other projects that were themselves at the front.

There are some issues that I want to address before the 0.8 and 1.0 releases, but I'm hoping that they're all internal. As a consequence of that, I'm also hoping that this documentation will only minimally change as clj-peg moves toward a 1.0 release. I'm still open to feedback, but at version 0.6, the details are starting to solidify. Please contact me if you have anything that I can help with. While I don't pretend to have as much time as I'm filling, I do try to care.

Good luck and don't be afraid to share your grammars.